Journal of Computer and System Sciences 76 (2010) 390–402

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Information flow analysis of scientific workflows

Ping Yang [a], Shiyong Lu [b,*], Mikhail I. Gofman [a], Zijiang Yang [c]

[a] Computer Science Dept., Binghamton University, Binghamton, NY 13902, United States
[b] Computer Science Dept., Wayne State University, Detroit, MI 48202, United States
[c] Computer Science Dept., Western Michigan University, Kalamazoo, MI 49008, United States

ARTICLE INFO

Article history:
Received 15 March 2009
Received in revised form 18 August 2009
Available online 22 November 2009

Keywords:
Information flow analysis
Scientific workflows
Hierarchical state machines

ABSTRACT

Recently, scientific workflows have emerged as a platform for automating and accelerating data processing and data sharing in scientific communities. Many scientific workflows have been developed for collaborative research projects that involve a number of geographically distributed organizations. Sharing of data and computation across organizations in different administrative domains is essential in such a collaborative environment. Because of the competitive nature of scientific research, it is important to ensure that sensitive information in scientific workflows can be accessed by and propagated to only authorized parties. To address this problem, we present techniques for analyzing how information propagates in scientific workflows. We also present algorithms for incrementally analyzing how information propagates upon every change to an existing scientific workflow.

1. Introduction

Today, scientists use scientific workflows to integrate, structure, and orchestrate a wide range of local and remote heterogeneous services and applications to perform various *in silico* experiments to produce scientific discoveries [27,20,19,37]. As a result, scientific workflows have become the de facto cyberinfrastructure upper-ware for e-Science [26]. Many scientific workflows have been developed for collaborative research projects that involve a number of geographically distributed organizations. Sharing of data and computation across organizations in different administrative domains is essential in such a collaborative environment. Because of the competitive nature of scientific research, it is important to ensure that sensitive information in scientific workflows can be accessed by and propagated to only authorized parties.

Let us consider a collaborative brain disorder research project that is conducted by a collaboration among two hospitals H_1 and H_2, a medical researcher R, and a computer scientist C. A typical medical scenario for using a scientific workflow is shown in Fig. 1. This figure shows various principals, datasets, and software tools in the system. In the figure, an oval represents a principal within the system; an arrow represents the direction of information flow between two principals; a square box represents a piece of data that is flowing; and a double oval represents a trusted software program. Each principal defines its own information flow policy which specifies a set of principals that can access its data. In our example, each patient p has two pieces of data: Positron Emission Tomography (PET) data that can be retrieved from hospital H_1, and functional Magnetic Resonance Imaging (fMRI) data that can be retrieved from hospital H_2. Each hospital limits the data only to the patient and the hospital itself. This is represented by $\{p, H_1\}$ and $\{p, H_2\}$, respectively. The program "Retrieve Data" is a trusted program which is run by the hospital to anonymize all personal information in the data and provide the anonymized data to researcher R (labeled with $\{p, R\}$). Researcher R can invoke a neuroimaging analysis workflow W to

* Corresponding author.
E-mail addresses: pyang@cs.binghamton.edu (P. Yang), shiyong@wayne.edu (S. Lu), mgofman1@binghamton.edu (M.I. Gofman), zijiang.yang@wmich.edu (Z. Yang).

0022-0000/$ – see front matter
doi:10.1016/j.jcss.2009.11.002

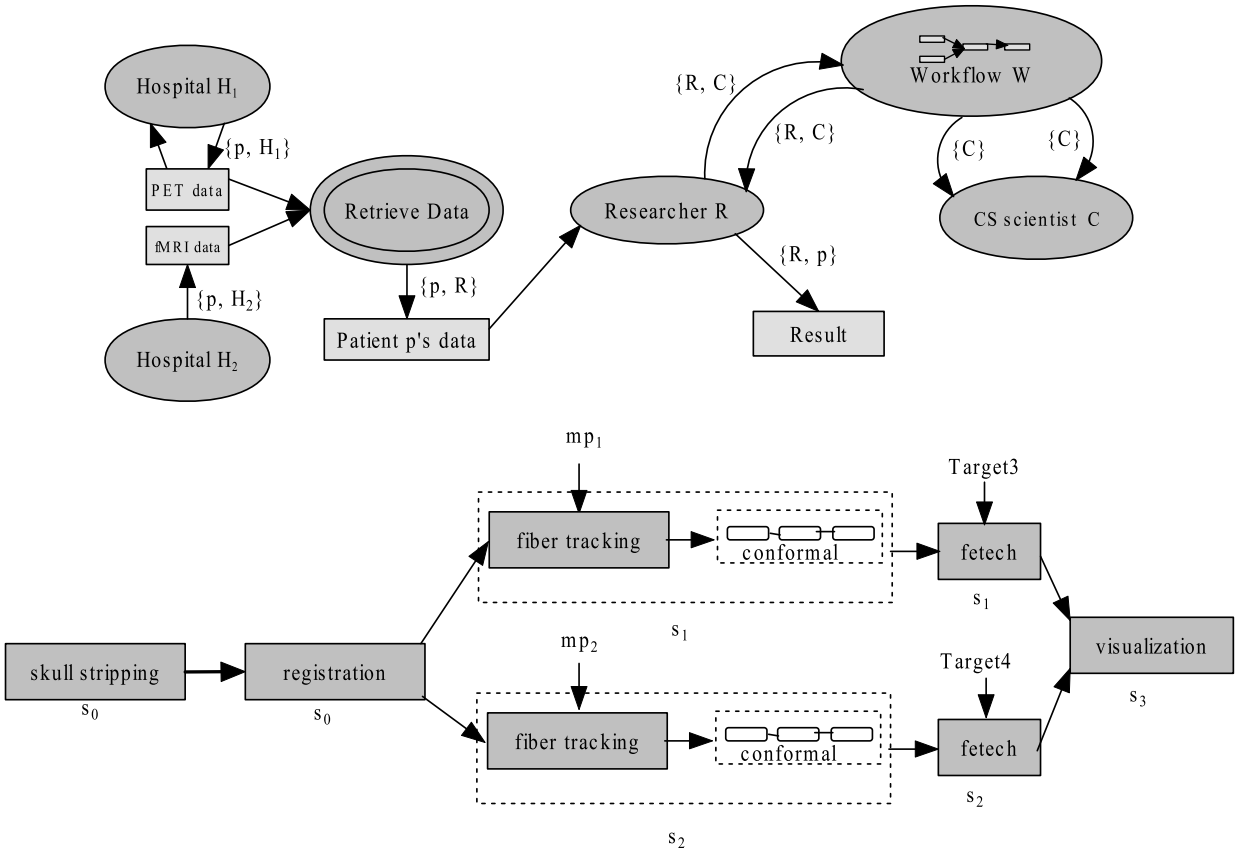


Fig. 1. A medical scenario for using scientific workflows.

identify each patient p's fiber tract pattern and abnormal cortical regions. In workflow W, arrows represent data channels and boxes represent workflow tasks. Workflow W consists of three tasks: *skull stripping*, *registration*, and *fiber tracking*. $S_0, \ldots, S_3$ specify the hosts at which workflow tasks are executed. In addition, a subworkflow for conformal mapping is reused in two branches of the same workflow. The execution of the workflow also needs the interaction from a computer scientist who specifies appropriate parameters. Finally, the result of the study is only readable to the patient and R. This is achieved by label $\{R, p\}$.

Based on the information flow policy prescribed by hospital H_1, a mechanism is needed to ensure that a patient's PET data, which is released from hospital H_1, can be accessed by and propagated to only authorized parties: hospital H_1, the patient herself, and researcher R. The PET data should not be accessed by or propagated to unauthorized parties such as hospital H_2. However, consider the following scenario. First, the PET data for a patient p stored in H_1 is retrieved via the *Retrieve Data* program and released to researcher R. Second, R executes workflow W. After the execution of tasks *skull stripping* and *registration*, the PET information is propagated to the first branch of the workflow. Third, the *fiber tracking* task in the first branch gets executed and the PET information is further propagated to the *conformal* subworkflow. Finally, suppose one of the tasks in the *conformal* subworkflow writes a file at a site that is owned by hospital H_2, then a violation of the information flow policy prescribed by hospital H_1 occurred: the PET information is indirectly propagated to hospital H_2. Such a violation can be hard to be detected manually due to the complexity of information flows among and within workflow tasks, particularly for large-scale hierarchical scientific workflows.

To address the above problem, we present information flow analysis techniques for analyzing how information propagates in scientific workflows. Our analysis techniques deal with both explicit and implicit information flows. Further, workflows tend to evolve over time and it would be inefficient to perform information flow analysis from scratch upon every small change to the structure of a workflow. Incremental analysis is useful in situations where small changes to the workflow lead to small or no changes to the analysis results. In this paper, we present an algorithm for incrementally analyzing information flows whenever a change is made to the structure of a workflow. We have also developed a prototype system, called *Infoflow Analyzer*, to validate and demonstrate our approaches. Although we present our analysis techniques in the context of scientific workflows, such techniques are also applicable to business workflows.

The rest of the paper is organized as follows. Section 2 provides an overview of hierarchical state machines. In Section 3, we present techniques for analyzing how information propagates in scientific workflows. We have also presented an algo-

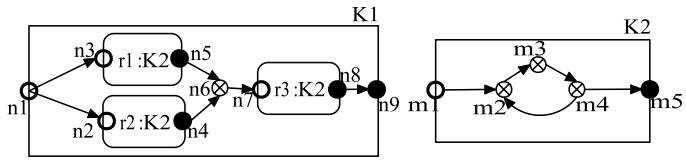


Fig. 2. A sample hierarchical state machine.

rithm for incrementally performing information flow analysis for evolving scientific workflows. The implementation details are given in Section 4. The related work and our concluding remarks appear in Sections 5 and 6, respectively.

2. Background: Hierarchical state machines

A Hierarchical State Machine (HSM) [6] K is a tuple $\langle K_1, \ldots, K_n \rangle$ of modules, in which each module K_i has the following components:

1. A finite set N_i of *states*, including a set of *entry states* I_i, a set of *exit states* O_i, and a set of *internal states* E_i that are neither entry states nor exit states.
2. A finite set B_i of *sub-modules*. The sets N_i and B_i are pairwise disjoint.
3. An indexing function $Y_i : B_i \mapsto \{i+1 \ldots n\}$, which maps each sub-module b of K_i ($b \in B_i$) to j with $j > i$. If $Y_i(b) = j$, then b is a reference to the definition of module K_j. Each pair (b, u) with $u \in I_j$ is called *a call* of K_i and each pair (b, v) with $v \in O_j$ is called *a return* of K_i.
4. A transition edge relation E_i of the form $u \xrightarrow{G,A} v$ where the source u is either a state or a return of K_i, the sink v is either a state or a call of K_i, G is a conditional guard, and A is an action.

Intuitively, a *call* of K_i is an entry state of a sub-module of K_i and a *return* of K_i is an exit state of a sub-module of K_i. Transitions are edges connecting states and modules with one another. A guard G over a transition specifies the condition under which the transition can be performed, and an action A is a sequence of variable assignments.

In an HSM, a state can be an ordinary state or a superstate, which is an HSM itself. An HSM K can be "flattened" to a finite state machine by recursively substituting each reference of K with the corresponding sub-module. Because the references of the same sub-module can reside in different modules, each module can appear in a number of different contexts. It has been shown in [6] that flattening may cause exponential blow-up, especially when there are many references pointing to the same module. A module is called a *top-level module* if it does not have parent modules. All references in the hierarchical state machine should form an acyclic graph.

Fig. 2 gives an example of an HSM. We use squares to denote module definitions and round-corner rectangles to denote module references. The entry states, exit states, and internal states are denoted by $\circ$, $\bullet$, and $\otimes$, respectively. Guards and actions are omitted in the figure. There are two modules: K_1 and K_2. K_1 is a top-level module that contains three sub-modules r_1, r_2, r_3, all of which are references to K_2, i.e., $Y_1(r_1) = Y_1(r_2) = Y_1(r_3) = 2$. K_1 has one entry state n_1 and one exit state n_9. (r_1, m_1), (r_2, m_1), and (r_3, m_1) are the calls of K_1. (r_1, m_5), (r_2, m_5), and (r_3, m_5) are the returns of K_1. K_2 is a module with five states and five transitions.

3. Information flow analysis of scientific workflows

This section presents information flow analysis techniques for scientific workflows and an algorithm for incrementally performing information flow analysis for evolving scientific workflows. We consider the workflows with the following control flow patterns: sequence, exclusive choice, parallel split, synchronization, simple merge, and condition. We assume that the permission for executing each workflow task is obtained from some appropriate access control mechanisms. We also assume that the source code of workflow tasks is available.

3.1. Explicit information flow analysis

We consider hosts as principals. Objects are system resources such as files and databases. Each object has an object ID. Object O in host h is specified as $h : O$. Each host h has a *host information flow policy*, $access(h)$, which specifies the set of principals who can access the objects in h. This policy can be overruled by an *object information flow policy*, $access(O)$, which specifies the set of hosts to which the information (i.e., the content) of O can flow. Information flows from an object O_1 to an object O_2 if information stored in O_1 is transferred to O_2 through a sequence of operations such as assignment statements, file reading and writing, I/O operations, and parameter passing. Given an information flow policy $access(O) = \{h_1, \ldots, h_n\}$, a workflow violates this policy if there exists an object O' in a host other than $h_1, \ldots, h_n$ such that information can flow from O to O'.

Formal modeling of scientific workflows. Simple Conceptual Unified Flow Language (SCUFL) [29] is an XML-based workflow specification language. Java BeanShell script [1] can be embedded in SCUFL to implement tasks in scientific workflows. In this paper, we consider scientific workflows specified using SCUFL and BeanShell script. For clarity of presentation and illustration purpose, we use an abstract syntax of Java BeanShell to illustrate our information flow analysis techniques. Let w be a workflow, *task* be an atomic task, *ports* be a set of ports, *prog* be a program, *vdecl* be a set of variable declarations, *pdecl* be a set of procedure declarations, *stmt* be a set of statements, and *cond* be a set of conditions. Also, let $x_1, \ldots, x_n$, $y_1, \ldots, y_n$, f and fp range over variables, p range over procedure names, and n range over constants. The core language syntax in its abstract form is given below:

$$w ::= task \mid w \xrightarrow{o,i} w$$

$$task ::= ports \triangleright prog \triangleright ports$$

$$prog ::= pdecl; end\_prog \mid pdecl; prog$$

$$pdecl ::= \text{proc}\, p(x_1, \ldots, x_n)\{vdecl; stmts\}$$

$$stmts ::= stmt; end\_stmt \mid stmt; stmts$$

$$stmt ::= x := y \mid x := n \mid x = fopen(f) \mid x = fread(fp) \mid fwrite(x, fp) \mid fclose(fp)$$
$$\mid \textit{if cond then stmts else stmts} \mid \text{call}\, p(y_1, \ldots, y_n) \mid \text{while}\ cond\ \text{do}\ stmts$$

$w_1 \xrightarrow{o,i} w_2$ specifies a data channel connecting the output port o of w_1 and the input port i of w_2. Each atomic task contains a set of input ports $\{i_1, \ldots, i_n\}$, a program implementing the functionality of the task, and a set of output ports $\{o_1, \ldots, o_m\}$. A program consists of a sequence of procedures separated by ";". *end_prog* signals the end of the program. Each procedure is of the form $proc\, p(x_1, \ldots, x_n)\{vdecl; stmts\}$ where *vdecl* is a local variable declaration of the form $var\, x_1, \ldots, x_n$ and *stmts* is a sequence of statements. $x = y$ and $x = n$ represent assignment of a variable y to x and assignment of a constant n to x, respectively. *fopen*, *fread*, and *fwrite* represent opening, reading, and appending an object, respectively. *if cond then* $stmt_1$ *else* $stmt_2$ is a conditional statement and *while cond do stmt* is a while loop. *call* $p(y_1, \ldots, y_n)$ represents the invocation of procedure $p(x_1, \ldots, x_n)$ with real parameters $y_1, \ldots, y_n$.

We propose the Hierarchical State Machine for Scientific Workflows (HSMSW), that refines the HSM to target the modeling of scientific workflows and the information flow analysis. HSMSW extends HSM with variable scoping, which enables us to model languages with nested blocks and different tasks that use the same name to represent different variables. A state of HSMSW is a set of set of variable assignments, each of which records the variable assignments propagated through one path. HSMSW assembles modules using transitions that connect exit states of one module with entry states of another module. Each transition is labeled with an action and a guard. The outgoing transitions of a module are either "and" transitions or "or" transitions, which models parallel split and exclusive choice, respectively. Similarly, the incoming transitions of a module are either "and" transitions or "or" transitions, which model synchronization and simple merge, respectively. Note that, our information flow analysis technique reports violation of an information flow property if there exists at least one path that violates the property. As a result, "and" and "or" transitions are handled in the same manner in the analysis.

Each atomic task in the workflow is modeled using an HSMSW specifying the internal structure of the task. Specifically, each procedure in the atomic task is modeled as an HSMSW. Local variables of a procedure are local variables of the corresponding HSMSW. The invocation of a procedure is modeled as a reference to the HSMSW of the procedure. Each occurrence of an atomic task is modeled as a reference to the corresponding module of the task. The input and output ports in a scientific workflow are modeled using entry and exit states in HSMSW, respectively. Data channels between two workflow tasks T_1 and T_2 are modeled as transitions between the two HSMSWs that model T_1 and T_2. A composite task (i.e., a task that contains sub-tasks) is modeled as a module with all sub-tasks modeled as modules in HSMSW.

Algorithms 1 and 2 give a procedure for translating an atomic task, specified using the abstract language defined above, into HSMSW. While statement is handled in a manner similar to the conditional statement and is not shown in the algorithm. Each variable in HSMSW does not contain its real value, e.g., content read from an object. Instead, the value of each variable v in HSMSW is either an object ID (when evaluating "$fp = fopen(f)$"), or a set of object IDs whose information can potentially flow to v.

Because a scientific workflow may be executed multiple times with different input datasets and parameter values and multiple workflow runs may access the same object, we associate each object O with a set $pastflow(O)$. $O_i \in pastflow(O)$ iff information has flown from O_i to O in previous workflow execution. $pastflow(O)$ is $\emptyset$ when object O is initially created. Below, we use an example to illustrate how $pastflow(O)$ is updated as a result of workflow execution. Suppose that $pastflow(O_1) = pastflow(O_2) = \emptyset$ before a workflow W_1 is executed and information of an object O_1 is transferred to object O_2 during the execution of W_1. As a result, $pastflow(O_2) = \{O_1\}$. Next, workflow W_2 is executed, which reads from O_2 and writes its content to O_3. To keep track of information propagation, $pastflow(O_2)$ is read as well so that we know O_2 contains the information originated from O_1. After W_2 finishes its execution, $pastflow(O_3) = \{O_1, O_2\}$, indicating that object O_3 contains the information that originated from both O_1 and O_2.

The top-level function *CreateHSMSW*(*prog*) in Algorithm 1 is used to construct an HSMSW for a program *prog*. For each procedure $proc\, p(x_1, \ldots, x_n)\{vdecl, stmts\}$ in the program, an HSMSW K_p is constructed (lines 2–9, Algo-

Algorithm 1 Algorithm for constructing *HSMSW* from an atomic task.

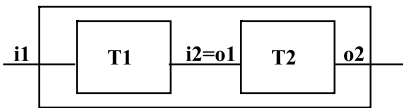
```
1: procedure CreateHSMSW(prog)
2: let prog = proc p(x_1, ..., x_n){vdecl; stmts}; prog'
3: Create a module K_p, an entry state t and an exit state t_e
4: Local variables of K_p = variables in vdecl
5: if prog' == end_prog then
6:    return AddMulStmts(stmts, t, t_e)
7: else
8:    return AddMulStmts(stmts, t, t_e) ∪ CreateHSMSW(prog')
9: end if
10: procedure AddMulStmts(stmts, t, t_e)
11: let stmts = stmt; stmts'
12: if stmt is "if cond then stmt_1 else stmt_2" then
13:    if stmts == end_stmt then
14:       return AddMulStmts(stmt_1, t, t_e) ∪ AddMulStmts(stmt_2, t, t_e)
15:    else
16:       Create a state t_1
17:       K_1 = AddMulStmts(stmt_1, t, t_1)
18:       K_2 = AddMulStmts(stmt_2, t, t_1)
19:       K_3 = AddMulStmts(stmts', t_1, t_e)
20:       return K_1 ∪ K_2 ∪ K_3
21:    end if
22: else
23:    if stmts == end_stmt then
24:       return AddSingleStmt(stmt, t, t_e)
25:    else
26:       Create a state t_1
27:       return AddSingleStmt(stmt, t, t_1) ∪ AddMulStmts(stmts', t_1, t_e)
28:    end if
29: end if
```

Algorithm 2 Algorithm for constructing *HSMSW* from an atomic task (Cont.).

```
1: procedure AddSingleStmt(stmt, t, t_1)
2: Create a state t_1
3: switch (stmt){
4:    case "x = y":
5:       return {t --x:=y--> t_1}
6:    case "x = n":
7:       return {t --x:=∅--> t_1}
8:    case "fp = fopen(f)":
9:       return {t --fp:=f--> t_1}
10:   case "x = fread(fp)":
11:      return {t --x:=pastflow(*fp)∪{*fp}, flow_w(*fp):=pastflow(*fp)--> t_1}
12:   case "fwrite(x, fp)":
13:      return {t --flow_w(*fp):=flow_w(*fp)∪x--> t_1}
14:   case "call p(y_1, ..., y_n)":
15:      constructs a reference R that refers to module K_p
16:      return {t --x_i:=y_i--> entry(R)} ∪ {exit(R) --ε--> t_1}
17: }
```

rithm 1). *AddMulStmts*$(stmts, t, t_e)$ is used to construct K_p from a sequence of statements *stmts* of p, an entry state t of K_p, and an exit state t_e of K_p. Lines 12–21 of Algorithm 1 construct HSMSWs from a sequence of statements *if* (C) *then* $stmt_1$ *else* $stmt_2; stmts'$. We first construct two HSMSWs with an entry state t and an exit t_1 from $stmt_1$ and $stmt_2$, respectively. We then construct an HSMSW with entry state t_1 and exit state t_e from $stmts'$.

AddSingleStmt$(stmt, t, t_1)$ in Algorithm 2 is used to generate transitions with source state t and target state t_1, from a statement *stmt* that is neither a conditional nor a while statement. Note that if one path of a workflow violates an information flow property, the workflow violates this property. Therefore, we consider all paths in the HSMSW regardless of the conditions in the transitions. As a result, the conditions are omitted during the construction of HSMSW. When $fp = fopen(f)$ is processed, the ID of the object f is assigned to fp (lines 8 and 9, Algorithm 2). When processing $x = fread(fp)$, x is assigned with $pastflow(*fp) \cup \{*fp\}$, where $*fp$ is the value of fp, i.e., the object ID stored in fp (lines 10 and 11, Algorithm 2). In addition, a variable $flow_w(*fp)$ is created and is assigned $pastflow(*fp)$ initially. $flow_w(f)$ is used to keep track of a set of object IDs from which information can potentially flow to f if workflow W gets executed. *fwrite* is similarly handled. *fclose*



T_1: input - i_1, output - o_1	T_2: input - i_2, output - o_2
$var\, fp_1, x, y, fp_2$	$var\, fp_1, x, y, fp_2;$
$a_1 : fp_1 = fopen(i_1);$	$b_1 : fp_1 = fopen(i_2);$
$a_2 : x = fread(fp_1);$	$b_2 : x = fread(fp_1);$
$a_3 : y := x;$	$b_3 : y := x;$
$a_4 : fclose(fp_1);$	$b_4 : fclose(fp_1);$
$a_5 : fp_2 = fopen(h_2 : f_2);$	$b_5 : fp_2 = fopen(h_3 : f_3);$
$a_6 : fwrite(y, fp_2);$	$b_6 : fwrite(y, fp_2);$
$a_7 : fclose(fp_2);$	$b_7 : o_2 := y;$
$a_8 : o_1 :=$ "$h_2 : f_2$";	$b_8 : fclose(fp_2);$

Fig. 3. Description of a sample workflow.

does not change the state. Lines 14–16 of Algorithm 2 describe how we generate transitions from procedure invocation. Every time a procedure invocation *call* $p(y_1, \ldots, y_n)$ is encountered, a reference R is created that points to module K_p. Two transitions $t \xrightarrow{x_i := y_i} entry(R)$ and $exit(R) \xrightarrow{\epsilon} t_1$ are constructed, where $entry(R)$ and $exit(R)$ are the entry and the exit states of R, respectively, and ϵ indicates that no action is performed.

Analysis algorithm. Every time a scientific workflow is constructed, a corresponding HSMSW is generated. Each task in a scientific workflow may have parameters whose values are provided by the user who executes the workflow. Once a user provides the input parameter values, the parameter values are passed to the HSMSW constructed and the static information flow analysis is performed to detect potential information leaks. Let $host(h : o) = h$. We say that a workflow W *conforms to the information flow policy* if and only if there does not exist a transition $s \xrightarrow{flow_w(*f) := flow_w(*f) \cup x} s'$ in the HSMSW of W such that $host(*f) \notin access(O)$ for some $O \in value(x)$.

The analysis algorithm is based on the reachability computation. First, the algorithm chooses a state s whose data is available and computes a set of states reachable from s. Let $s_1 = \{s_{11}, \ldots, s_{1n}\}, \ldots,$ and $s_m = \{s_{m_1}, \ldots, s_{mk}\}$ be m states, and transitions $s_i \xrightarrow{\alpha_i} t$ $(1 \leqslant i \leqslant m)$ be all incoming transitions of t. Then t is computed as $\{s_{11} \cup eval(s_{11}, \alpha_1), \ldots, s_{mk} \cup eval(s_{mk}, \alpha_m)\}$, where $eval(s, \alpha)$ evaluates the action α on state s. For example, let $s = \{\{x = 1, flow_w(h_1 : f_1) = \{h_2 : f_2\}\}, \{flow_w(h_1 : f_1) = \{h_3 : f_3\}\}\}$ be a state and let $s \xrightarrow{x:=2, flow_w(h_1:f_1) := flow_w(h_1:f_1) \cup \{h_1:f_2\}} t$ be a transition. Then t is computed as $\{\{x = 2, flow_w(h_1 : f_1) = \{h_2 : f_2, h_1 : f_2\}\}, \{x = 2, flow_w(h_1 : f_1) = \{h_3 : f_3, h_1 : f_2\}\}\}$.

Repeat the above process until it "gets stuck" at a state. A state s is called a *stuck state* if s waits for inputs from other states. When a stuck state is encountered, the algorithm finds another state whose data is available and perform reachability computation. The algorithm terminates if a violation to the information flow policy is detected or all states whose data are available have been processed. Once our algorithm reports a violation, the administrators will be prompted to revise the workflow. The workflow is executed if no violation is detected in the analysis.

Consider the workflow W in Fig. 3, which is a composite task containing two sub-tasks T_1 and T_2. T_1 and T_2 execute in sequential order and the output of T_1 acts as the input to T_2. T_1 reads from input i_1 and appends the content read to file $h_2 : f_2$. T_2 reads from i_2 and writes to $h_3 : f_3$. The HSMSWs of W, T_1, and T_2 are given in Figs. 4(a), 4(b), and 4(c), respectively.

Assume that $pastflow(h_1 : f_1) = pastflow(h_2 : f_2) = \emptyset$ before W executes. Also, assume that $access(h_1 : f_1) = \{h_2, h_3\}$ and $access(h_2 : f_2) = \{h_3\}$, which specify that only organizations h_2 and h_3 can access file f_1 of h_1 and only organization h_3 can access file f_2 of h_2, respectively. Suppose that a user executes the workflow W with input $h_1 : f_1$, then i_1 in Fig. 4(b) is assigned $h_1 : f_1$ and state $s_1 = \{\{i_1 = h_1 : f_1\}\}$. After $h_1 : f_1$ is read (lines a_1 and a_2), $x = pastflow(h_1 : f_1) \cup \{h_1 : f_1\} = \{h_1 : f_1\}$, $flow_w(h_1 : f_1) = pastflow(h_1 : f_1) = \emptyset$, and $s_3 = \{\{x = \{h_1 : f_1\}, flow_w(h_1 : f_1) = \emptyset\}\}$. After x is assigned to y (line a_3), $y = \{h_1 : f_1\}$. When $fwrite(y, h_2 : f_2)$ is evaluated (lines a_5 and a_6), $flow_w(h_2 : f_2) = \{h_1 : f_1\}$. Because $h_2 \in access(h_1 : f_1)$, the information flow policy is not violated and hence the content of y can be written to $h_2 : f_2$. Similarly, after processing T_2, $flow_w(h_3 : f_3) = \{h_1 : f_1, h_2 : f_2\}$. Since no information flow violation is detected, the workflow can be executed. Now, suppose that we change $access(h_1 : f_1)$ to $\{h_2\}$. Then T violates this policy because there exists a transition $s_{12} \xrightarrow{flow_w(h_3:f_3) := flow_w(h_3:f_3) \cup y} s_{13}$ such that $y = \{h_1 : f_1, h_2 : f_2\}$ and $host(h_3 : f_3) \notin access(h_1 : f_1)$.

$pastflow(O)$ is updated during the execution of the scientific workflow. For each variable x in the workflow, we associate x with an auxiliary variable aux_x, which has the same scoping as x. For each statement s, we insert the corresponding action generated from Algorithm 3 (with x replaced with aux_x) immediately after the statement. For example, $aux_x := aux_y$ is inserted after the statement $x = y$. After the workflow finishes the execution, $aux_{flow_w(*f)}$ is written back to $pastflow(*f)$.

Below, we prove the correctness of our algorithm.

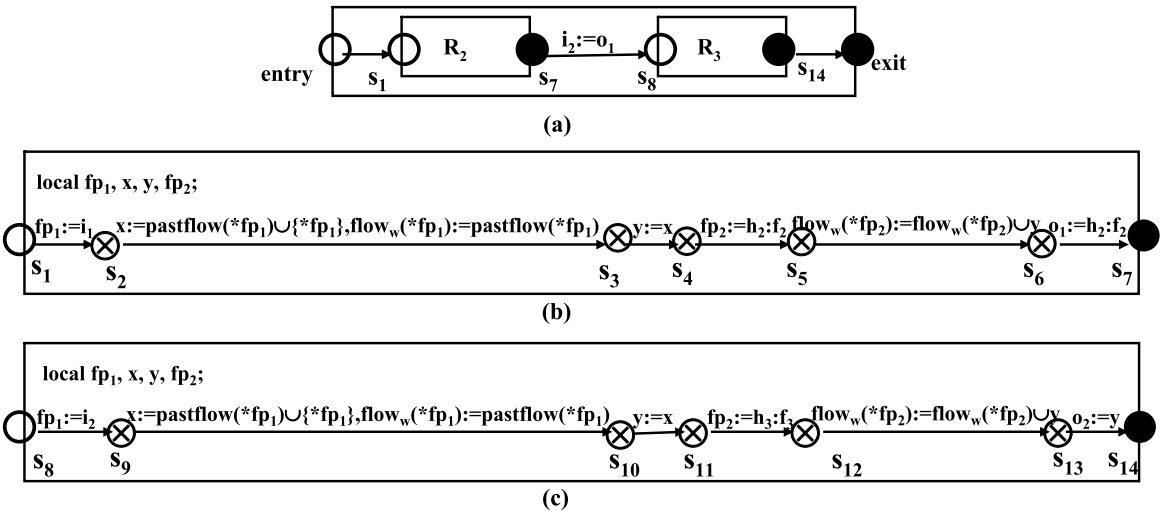


Fig. 4. (a) The HSMSW K_1 modeling workflow W. (b) The HSMSW K_2 modeling task T_1. (c) The HSMSW K_3 modeling task T_2. R_2 and R_3 in (a) are references to K_2 and K_3, respectively.

Theorem 3.1. *Given a workflow W, the HSMSW $HSMSW_w$ of W, and an information flow policy $access(O) = \{h_1, \ldots, h_n\}$, the following statements hold*: (1) *W conforms to $access(O)$ if and only if there does not exist a transition $s \xrightarrow{flow_w(*f):=flow_w(*f)\cup x} s'$ in the $HSMSW_w$ such that $O \in value(x)$ and $host(*f) \notin access(O)$.* (2) *For every variable v in $HSMSW_w$, $value(v)$ is the set of all objects from which the information flows to v.*

Proof. The theorem is proved by an induction on the size of W.

Base case: *$HSMSW_w$ has only one transition.* An information flow policy can be violated only when W contains at least one *fread* statement and one *fwrite* statement. This means that, only when $HSMSW_w$ contains at least two transitions, one for *fread* and one for *fwrite*, the policy can be violated. Our algorithm reports no violation when $HSMSW_w$ has only one transition. Therefore (1) holds.

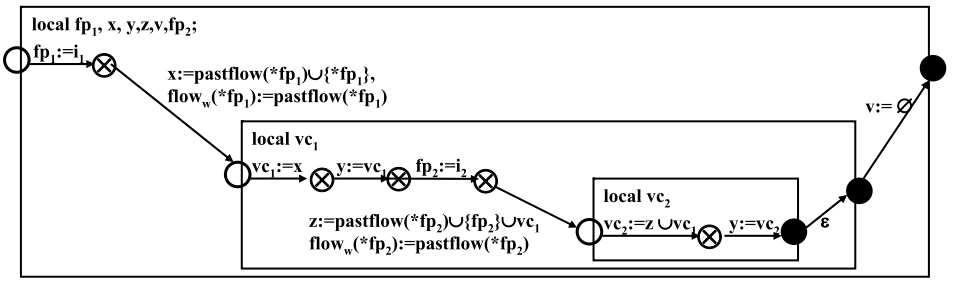
We now prove the base case for (2). If the transition corresponds to a statement that does not contain *fread* and *fwrite* (no information flows within the workflow), then $value(x) = \emptyset$ for every variable x in $HSMSW_w$ and hence (2) holds. If the transition corresponds to statement $x = fread(fp)$ (information flows from the object pointed by fp to x), then $value(x) = \{*fp\} \cup pastflow_w(*fp)$ and hence (2) holds. If the transition corresponds to statement $fwrite(x, fp)$, this means that the content of variable x is written to the object pointed by fp. Since no object is read before the *fwrite* statement, no information flows from any object to x and hence no information flows to the object pointed by fp. Our algorithm computes $value(x) = flow_w(*fp) = \emptyset$, which is consistent with the above.

Induction step: Assume that (1) holds for all workflows of size k. We now prove that when the size of W is $k+1$, (1) holds as well. Let *stmt* be the last statement of W. If $W \setminus \{stmt\}$ does not conform to $access(O)$, then W also does not conform to $access(O)$. By induction hypothesis, there exists a transition $s \xrightarrow{flow_w(*f):=flow_w(*f)\cup x} s'$ in the $HSMSW_w$ such that $O \in value(x)$ and $host(*f) \notin access(O)$, and hence (1) holds. Otherwise, if *stmt* is not $fwrite(x, fp)$, then *stmt* will not cause $access(O)$ to be violated and hence W also conforms to $access(O)$. By induction hypothesis, there does not exist a transition $s \xrightarrow{flow_w(*f):=flow_w(*f)\cup x} s'$ in the $HSMSW_w$ such that $O \in value(x)$ and $host(*f) \notin access(O)$. Thus (1) holds. If *stmt* is $fwrite(x, fp)$, then information of x flows to $*fp$. Our algorithm generates action $flow_w(*fp) = flow_w(*fp) \cup x$, which means that $*fp$ contains information of x as well as the set of objects from which the information previously flows to $*fp$. By induction hypothesis, $value(x)$ is the set of all objects from which the information flows to x. Therefore, $flow_w(*fp)$ contains all information flowed to $*fp$. In this case, $access(O)$ is violated if and only if information flows from O to x and the host that owns $*fp$ is not allowed to access O, i.e. $O \in value(x)$ and $host(*fp) \notin access(O)$. Thus (1) holds.

We now prove (2). If *stmt* is an assignment statement $y = x$, then after this assignment, y contains the information of x. The action in the corresponding transition is $y := x$, which means that $value(y) = value(x)$. By induction hypothesis, $value(x)$ is the set of all objects from which the information flows to x, and hence $value(y)$ contains all objects from which the information flows to y. Thus (2) holds. If *stmt* is $x = fread(fp)$, then the information flows from $*fp$ to x. The action in the corresponding transition is $x := pastflow(*fp) \cup \{*fp\}$, $flow_w(*fp) := pastflow(*fp)$. Because $pastflow(*fp)$ contains all objects from which information flows to $*fp$, x contains all objects from which information flows to x. Thus, (2) holds. Similarly, we can prove that (2) holds for other statements. □

input: i_1, i_2;
$var\ fp_1, x, y, z, v, fp_2$;
$fp_1 = fopen(i_1)$;
$x = fread(fp_1)$;
if $(x > 1)\{$
  $y = 1$;
  $fp_2 = fopen(i_2)$;
  $z = fread(fp_2)$;
  if $(z > 2)\ \ y = 2$;
  else $\ \ y = 3;\}$
$v = 1$;

(a)



(b)

Fig. 5. An example and the corresponding HSMSW for analyzing implicit information flow.

3.2. Implicit information flow analysis

Algorithm 1 deals with only explicit information flow. It does not detect implicit information flow resulting from conditional statements such as *if* $(x == 0)$ *then* $y = 0$; *else* $y = 1$. In this statement, y is assigned 0 if the value of x is 0, and 1 otherwise. From y's value, we can infer whether x is equal to 0 or not. As a result of execution of this statement, information of x flows to y implicitly.

We propose to make use of the variable scoping of HSMSW to detect implicit information flow. For every conditional statement c of the form *if cond then* $stmt_1$ *else* $stmt_2$, we construct an HSMSW module K that contains a new local variable vc_k, called *implicit flow variable*, to store the union of values of all variables in *cond*, denoted as *vars*(*cond*). The value of vc_k is then propagated to variables that appear on the left-hand side of actions (assignments) generated from statements contained in c. A nested conditional statement with nesting-depth n is modeled by an HSMSW with nesting-depth n. Once the HSMSW is constructed, we can directly use the analysis technique presented in Section 3.1 to analyze implicit information flow.

The algorithm for constructing an HSMSW for analyzing implicit information flow of a conditional statement is given in Algorithm 3. The HSMSW for while statement can be constructed similarly. The algorithm for constructing HSMSWs for other statements is similar to that in Algorithm 1. In order to analyze implicit information flow, one parameter *vc* is added to *AddMulStmts* which is used to propagate the value of the implicit flow variable. Given a statement $stmt; stmts'$ where *stmt* is a conditional statement of the form *if cond then* $stmt_1$ *else* $stmt_2$, $AddMulStmts(stmt; stmts', t, t_e, vc)$ constructs an HSMSW K from $stmt; stmts'$ with an entry state t and an exit state t' as follows. First, a local implicit flow variable vc_k is created (line 4) and a transition $t \xrightarrow{vc_k := vars(cond) \cup vc} t_1$ is generated which assigns $vars(cond) \cup vc$ to vc_k (line 5). Next, $AddMulStmts(stmt_1, t_1, t', vc_k)$ and $AddMulStmts(stmt_2, t_1, t', vc_k)$ are called, which propagate vc_k to the left-hand side of the actions generated through $stmt_1$ and $stmt_2$, respectively (lines 7 and 8). The value of the implicit flow variable is propagated to all actions except the action generated through *fopen* and the assignment $flow_w(*fp) := pastflow(*fp)$ generated through *fread*. Finally, an ϵ transition from t' to t_e is generated (line 9). The sequence of statements $stmt; stmts'$ where $stmts' \neq end\_stmt$ is handled similarly.

Fig. 5(b) gives the HSMSW generated for the example in Fig. 5(a). When the conditional statement *if* $(x > 1)$ *then* ... in Fig. 5(a) is evaluated, a new module is constructed and a local variable vc_1 is created which stores the value of x. The value of vc_1 is them propagated to y and z, which means that the information of x flows to y and z.

3.3. Incremental information flow analysis

If our information flow analysis algorithm detects potential information flow violation in a scientific workflow, the user who constructed the workflow will be prompted to revise the workflow. A user may also want to change the structure of a scientific workflow if she is not satisfied with the results produced by the workflow. Incremental information flow analysis is useful when structural changes of a workflow lead to small or no changes in the analysis results. In some cases, a complete re-analysis cannot be avoided, but in most cases, incremental analysis allows us to reuse the previous analysis results and perform analysis more quickly than a complete re-analysis. In this section, we present techniques for incremental information flow analysis of scientific workflows. The basic idea is to store the exit states of all modules of HSMSW constructed in the previous analysis and then incrementally update HSMSW and perform analysis.

We consider the following changes to the structure of a scientific workflow: adding, deleting, or replacing workflow tasks, and adding and deleting data channels. Every time a change is made to the structure of a scientific workflow, our incremental algorithm starts from the entry states of the top module of HSMSW generated in the previous analysis and inspects every module and transition to see if it is affected by the change. If so, the algorithm updates the corresponding

Algorithm 3 Algorithm for constructing an HSMSW from an atomic actor for implicit information flow control.

procedure *AddMulStmts*($stmt$; $stmts'$, t, t_e, vc)
let $stmt$ = "if *cond* then $stmt_1$ else $stmt_2$"
Create an HSMSW module K with entry state t and exit state t'
Create a new implicit flow variable vc_k and a state t_1
$Trans_1 = \{t \xrightarrow{vc_k := vars(cond) \cup vc} t_1\}$
if $stmts' = end\_stmt$ **then**
  $K_1 = AddMulStmts(stmt_1, t_1, t', vc_k)$
  $K_2 = AddMulStmts(stmt_2, t_1, t', vc_k)$
  return $K_1 \cup K_2 \cup Trans_1 \cup \{t' \xrightarrow{\epsilon} t_e\}$
else
  Create a state t_3
  $K_1 = AddMulStmts(stmt_1, t_1, t_3, vc_k)$
  $K_2 = AddMulStmts(stmt_2, t_1, t_3, vc_k)$
  $K_3 = AddMulStmts(stmts', t_3, t', vc)$
  return $K_1 \cup K_2 \cup K_3 \cup Trans_1 \cup \{t' \xrightarrow{\epsilon} t_e\}$
end if

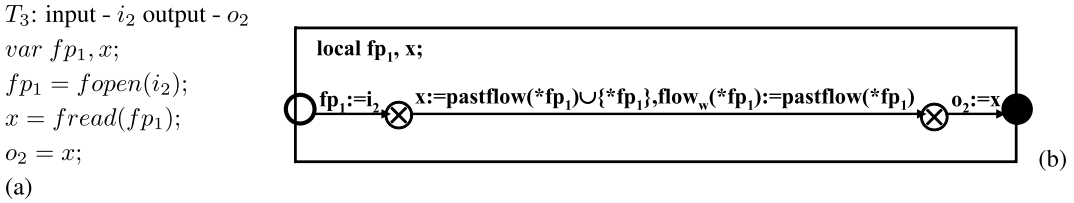


Fig. 6. (a) Specification of T_3; (b) the corresponding HSMSW of T_3.

modules and transitions. The algorithm for updating HSMSW is straightforward: If a task T is deleted from a workflow W, the corresponding reference in the HSMSW is deleted. If a new task T is added to a workflow W, an HSMSW module for T is constructed and a reference to this module is added correspondingly. Replacing a task T_1 with another task T_2 can be reduced to deleting T_1 and then adding T_2. If a data link is added to (or removed from) the workflow, the corresponding transition will be added to (or removed from) HSMSW.

After the user provides inputs to the scientific workflow, the incremental analysis is performed to detect potential information flow violation. The algorithm starts from the initial state and traverses HSMSW until a module or a connection channel that is affected by the change is encountered. Such a module or a connection channel is called an *affected point*. The algorithm then re-analyzes HSMSW from the affected point until a violation is detected or no more state is reachable (i.e., no violation is detected). The following optimizations can be applied when a module K_1 is replaced with another module K_2: (1) if the exit state of K_2 is the subset of that of K_1 and the previous analysis result is false (i.e., no violation is detected), then the algorithm returns false; and (2) if the exit state of K_2 is the superset of that of K_1 and the previous analysis result is true, then the algorithm returns true.

Below, we use the example in Fig. 3 to illustrate our incremental analysis technique. In Section 3, we have shown that this workflow violates the information flow policy $access(h_1 : f_1) = \{h_2\}$. During the analysis, we store the value of the exit state s_7 of R_2, which contains $\{flow_w(h_2 : f_2) = \{h_1 : f_1\}, o_1 = \{h_2 : f_2\}, \ldots\}$. Because the information flow policy of $h_1 : f_1$ is violated, the user is prompted to revise the workflow. Assume that the user replaces task T_2 with task T_3 in Fig. 6(a). Because T_1 does not change, R_2 does not change. Since T_2 is replaced with T_3, R_3 is replaced with a reference to the HSMSW module of T_3, i.e., the HSMSW shown in Fig. 6(b). Our incremental analysis algorithm then starts from s_7 and performs analysis using the technique given in Section 3. The analysis shows that the information flow policy $access(h_1 : f_1) = \{h_2\}$ is not violated.

4. Implementation and experimental results

We have developed an information flow analysis tool for scientific workflows, called *Infoflow Analyzer*, whose architecture is given in Fig. 7. Our Infoflow Analyzer was developed over Hermes [5], which is a tool for constructing a hierarchical reactive machine, an extension of the hierarchical state machine with variable scoping, and performing reachability analysis on the hierarchical reactive machine. In Hermes, the hierarchical reactive machine can either be drawn manually or be generated through an XML specification. Our Infoflow Analyzer differs from Herms as follows: (1) We extended the XML specification of the hierarchical reactive machine in Herms to support the specification of "and" and "or" transitions of HSMSW; (2) We implemented a translator for converting the workflow specification into the XML specification of the HSMSW; and (3) We implemented the information flow analysis algorithm developed in this paper which performs analysis on the XML specification of the HSMSW.

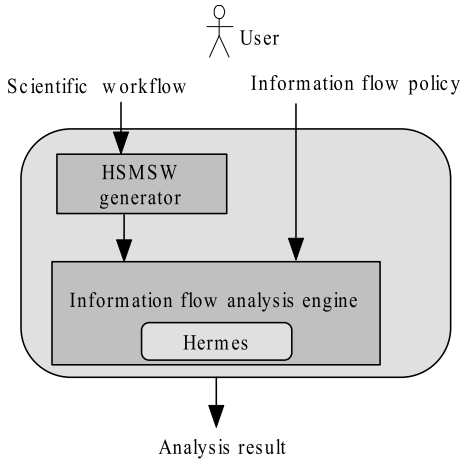


Fig. 7. The architecture of Infoflow Analyzer.

Fig. 8 gives the screenshots of HSMSW generated using Infoflow Analyzer for a workflow containing three tasks. M_1 and M_2 model tasks T_1 and T_2 in Fig. 3, respectively. Fig. 8(a) provides a top-level view of HSMSW, which allows users to provide parameter values and specify the information flow policy. Fig. 8(b) shows that the information flow policy has been violated by providing a detailed explanation of the cause and highlighting the transition that violates the policy.

Our prototype currently supports only a subset of Java BeanShell containing some basic file operations and assignment statements. Further development includes supporting a more expressive subset of Java BeanShell, including database operations.

Experimental results. We evaluate the scalability of our Infoflow Analyzer on workflows constructed by composing a number of modules corresponding to the task T_1 in Fig. 3 via connection channels. The number of tasks varies between 2000 (14K states) and 20 000 (140K states). Figs. 9(a) and 9(b) give the execution time and memory consumed by Infoflow Analyzer. All reported results were obtained on an Intel Pentium D machine with 1 GB of RAM running Linux 2.6.24. The figures show that it takes Infoflow Analyzer less than 100 seconds and less than 800 MB memory to analyze a workflow with 20 000 tasks. Obtaining results for more sophisticated real-world workflows is left as a topic for future work.

5. Related work

Much work has been done for modeling, analysis, and verification of workflows [2,3,12,33,30,17,16,25,14,10,9,11,34]. In this paper, we propose to control information propagation of scientific workflows based on *Hierarchical State Machines for Scientific Workflows* (*HSMSWs*). We exploit the hierarchical structure of *HSMSWs* to model scientific workflows and perform analysis directly on *HSMSWs* without flattening them into finite state machines, which avoids exponential blow-up caused by flattening. To the best of our knowledge, no information flow analysis techniques are proposed based on other formalisms.

While early scientific workflow effort demonstrated the capability of database management for supporting scientific workflows [4], several scientific workflow management systems (SWFMSs) have been developed over the past few years. The Kepler system [27] is a Java-based open source SWFMS. In Kepler, a scientific workflow is composed from components called *actors* and its execution is controlled by a computational model controller called *director*. Taverna [29] is another SWFMS targeted for life science. Currently, Taverna supports a repository of Web services for various bioinformatics data analysis and transformation. Taverna uses an XML-based workflow language called *SCUFL* for workflow representation with each component being either a Web service or a *processor* developed using Java BeanShell script. The Triana system [13] has a sophisticated graphical user interface for workflow composition and modification, including grouping, editing, and zooming functions. The VisTrails system [19] is developed to manage visualizations and is the first system that supports provenance tracking of workflow evolution in addition to tracking the data product derivation history. The Pegasus system [20] provides a framework which maps complex scientific workflows onto distributed grid resources. Artificial intelligence planning techniques are used in Pegasus for workflow composition. The Swift system [37] combines a novel scripting language called *SwiftScript* with a powerful runtime system to support the concise specification, and reliable and efficient execution, of large loosely coupled computations over grid environments. Finally, VIEW [24] system features an open and flexible service-oriented architecture, efficient provenance management using Semantic Web technologies, and advanced techniques for scientific data visualization. However, none of the above systems support information flow analysis.

The area of information flow analysis has received considerable attention. The lattice model of information flow was first proposed in [7] and [15]. Recently, a number of information flow control techniques have been developed for decentralized systems or Web services (e.g. [28,22,31,23,36]). However, none of these work is done in the context of scientific workflows or

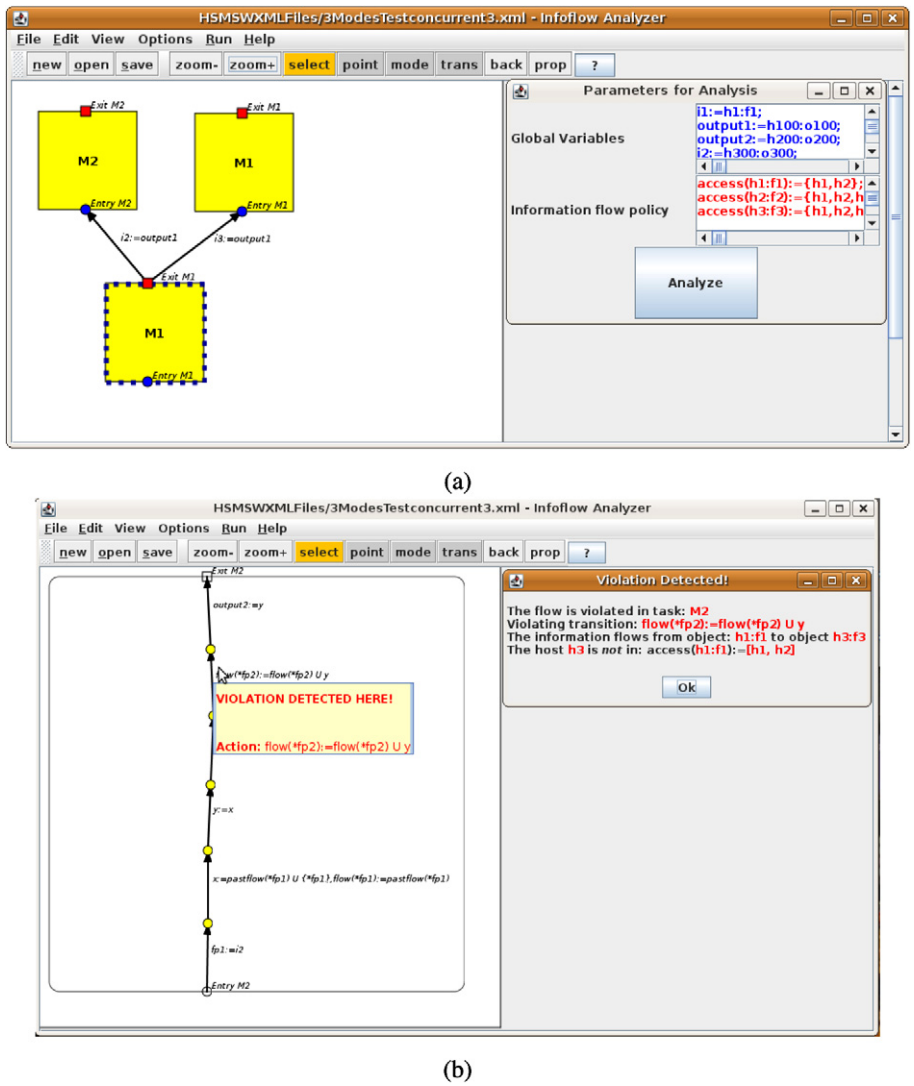


Fig. 8. The screenshots of Infoflow Analyzer for the example in Fig. 3.

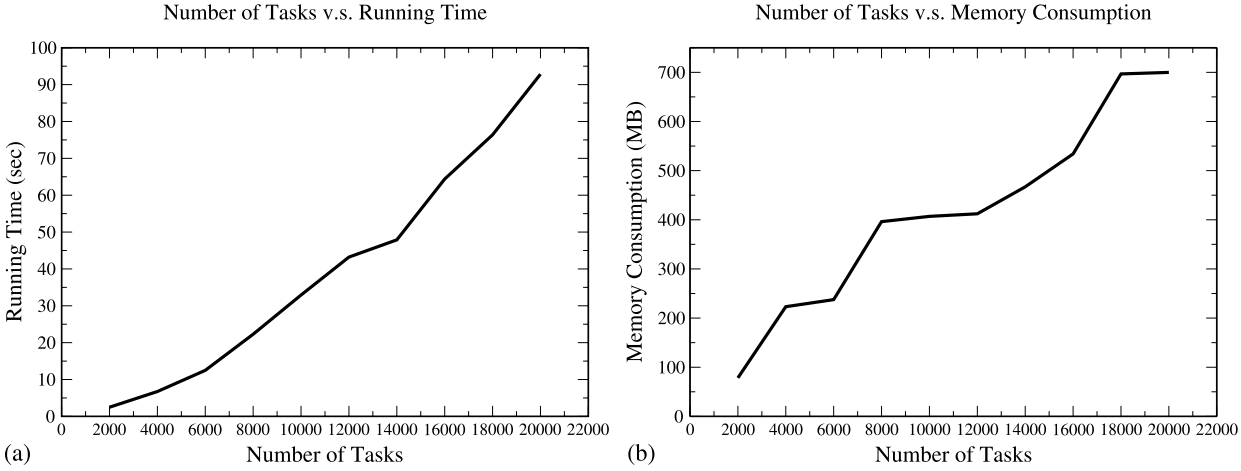


Fig. 9. The execution time and memory consumption of Infoflow Analyzer on workflows containing 2000–20 000 tasks.

uses hierarchical state machines to perform information flow analysis. Also, in addition to support information flow analysis, our framework also allows us to verify many other properties such as deadlock-freedom [35]. Guernic et al. [21] present an automata-based dynamic information flow analysis technique, while the information flow analysis technique presented in this paper is a static analysis technique. Information flow analysis has also been applied to programs (e.g. [32]), which checks if information may flow from a variable labeled "high" to a variable labeled "low" in the program. Our technique does not assume an order exists among variables, which allows us to specify a policy that prevents information from flowing among arbitrary sets of principals. Second, our analysis technique also accounts for dataflows formed from data channels that connect the input and output ports of workflow tasks.

Recently, the Kepler system extends its actor-oriented modeling framework with frames and templates by borrowing ideas from hierarchical state machines [8]. This approach seamlessly integrates control flows into a dataflow-based design paradigm without sacrificing the benefits of dataflows. Although Kepler provides a *concrete* hybrid model for designing and executing scientific workflows with both dataflows and control flow features, verification and information flow analysis are not part of the framework. In contrast, we aim at developing an *abstract* model for scientific workflows based on hierarchical state machines, providing a foundation for formal modeling and analysis of scientific workflows, including information flow analysis.

In [35], we have presented techniques for verifying and controlling information propagation of scientific workflows. However, [35] does not handle implicit information flow and does not consider incremental information flow analysis. The language and the control flow patterns we consider in this paper are also more expressive than those of [35]. For example, this paper considers synchronization, parallel split, and simple merge, while [35] does not. Further, no prototype is developed in [35].

6. Conclusions and future work

This paper presents information flow analysis techniques that control the information propagation in scientific workflows. We consider workflows with the following control flow patterns: sequence, exclusive choice, parallel split, synchronization, simple merge, and conditions. In order to perform formal analysis, we propose a *Hierarchical State Machine for Scientific Workflows* (HSMSW) and present information flow analysis techniques to control the information propagation in scientific workflows based on HSMSW. Our analysis techniques deal with both explicit and implicit information flows. We also present an efficient algorithm for performing information flow analysis incrementally upon every change to scientific workflows. We have developed a prototype system, called *Infoflow Analyzer*, to validate and demonstrate our approaches. For future work, we plan to extend our work to support scientific workflows with Web service components and scientific workflows running on cloud computing environments [18].

References

[1] Java BeanShell, http://www.beanshell.org/.
[2] W. Aalst, A. Hofstede, Verification of workflow task structures: A Petri-net-based approach, Inf. Syst. 25 (1) (2000) 43–69.
[3] N.R. Adam, V. Atluri, W. Kuang Huang, Modeling and analysis of workflows using Petri nets, J. Intell. Inf. Syst. 10 (2) (1998) 131–158.
[4] A. Ailamaki, Y.E. Ioannidis, M. Livny, Scientific workflow management by database management, in: Proc. of the International Conference on Statistical and Scientific Database Management, 1998, pp. 190–199.
[5] R. Alur, M. McDougall, Z. Yang, Exploiting behavioral hierarchy for efficient model checking, in: International Conference on Computer-Aided Verification, 2002.
[6] R. Alur, M. Yannakakis, Model checking of hierarchical state machines, in: Foundations of Software Engineering, 1998, pp. 175–188.
[7] D. Bell, L. LaPadula, Secure computer system: Unified exposition and multics interpretation, Technical Report ESD-TR-75-306, MITRE Corp., 1975.
[8] S. Bowers, B. Ludäscher, A. Ngu, T. Critchlow, Enabling scientific workflow reuse through structured composition of dataflow and control-flow, in: IEEE Workshop on Workflow and Data Flow for Scientific Applications, 2006.
[9] J. Chen, Y. Yang, Activity completion duration based checkpoint selection for dynamic verification of temporal constraints in grid workflow systems, Int. J. High Perform. Comput. Appl. 22 (3) (2008) 319–329.
[10] J. Chen, Y. Yang, A taxonomy of grid workflow verification and validation, Concurr. Comput.: Pract. Exper. 20 (4) (2008) 347–360.
[11] J. Chen, Y. Yang, Temporal dependency based checkpoint selection for dynamic verification of temporal constraints in scientific workflow systems, ACM Trans. Software Eng. Methodol., in press.
[12] Y. Choi, X. Zhao, K. Han, Hierarchical reachability analysis for workflow-nets, in: Conference on Computer Supported Cooperative Work in Design, 2006, pp. 556–561.
[13] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, I. Wang, Programming scientific and distributed workflow with Triana services: Research articles, Concurr. Comput.: Pract. Exper. 18 (10) (2006) 1021–1037.
[14] H. Davulcu, M. Kifer, C. Ramakrishnan, I. Ramakrishnan, Logic based modeling and analysis of workflows, in: PODS, 1998, pp. 25–33.
[15] D.E. Denning, A lattice model of secure information flow, Commun. ACM 19 (5) (1976) 236–243.
[16] M. Dumas, A.H.M. ter Hofstede, UML activity diagrams as a workflow specification language, in: UML, 2001, pp. 76–90.
[17] R. Eshuis, R. Wieringa, Verification support for workflow design with UML activity graphs, in: Proc. of the International Conference on Software Engineering, 2002, pp. 166–176.
[18] I. Foster, Y. Zhao, I. Raicu, S. Lu, Cloud computing and grid computing 360-degree compared, in: Proc. of the IEEE Grid Computing Environments, Austin, TX, 2008, pp. 1–10.
[19] J. Freire, C.T. Silva, S.P. Callahan, E. Santos, C.E. Scheidegger, H.T. Vo, Managing rapidly-evolving scientific workflows, in: International Provenance and Annotation Workshop, 2006, pp. 10–18.
[20] Y. Gil, E. Deelman, J. Blythe, C. Kesselman, H. Tangmunarunkit, Artificial intelligence and grids: Workflow planning and beyond, IEEE Intell. Syst. 19 (1) (2004) 26–33.

[21] G.L. Guernic, A. Banerjee, T. Jensen, D. Schmidt, Automata-based confidentiality monitoring, in: 11th Annual Asian Computing Science Conference, 2006.
[22] D. Hutter, M. Volkamer, Information flow control to secure dynamic web service composition, in: International Conference on Security in Pervasive Computing, 2006.
[23] P. Li, S. Zdancewic, Practical information-flow control in web-based information systems, in: Computer Security Foundation Workshop, 2005, pp. 2–15.
[24] C. Lin, S. Lu, Z. Lai, A. Chebotko, X. Fei, J. Hua, F. Fotouhi, Service-oriented architecture for VIEW: A visual scientific workflow management system, in: Proc. of the IEEE 2008 International Conference on Services Computing (SCC 2008), 2008, pp. 335–342.
[25] S. Lu, A.J. Bernstein, P.M. Lewis, Automatic workflow verification and generation, Theoret. Comput. Sci. 353 (1–3) (2006) 71–92.
[26] B. Ludäscher, Scientific workflows: Cyberinfrastructure for e-science, in: Pacific Neighborhood Consortium (PNC), 2007.
[27] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E.A. Lee, J. Tao, Y. Zhao, Scientific workflow management and the Kepler system: Research articles, Concurr. Comput.: Pract. Exper. 18 (10) (2006) 1039–1065.
[28] A.C. Myers, B. Liskov, A decentralized model for information flow control, in: SOSP, 1997, pp. 129–142.
[29] T. Oinn, et al., Taverna: lessons in creating a workflow environment for the life sciences: Research articles, Concurr. Comput.: Pract. Exper. 18 (10) (2006) 1067–1100.
[30] Y. Pan, Y. Tang, H. Ma, N. Tang, Workflow analysis based on fuzzy temporal workflow nets, in: CSCWD (Selected papers), 2005, pp. 545–553.
[31] N. Ravi, M. Gruteser, L. Iftode, Information flow control for location-based services, in: 3rd Annual International Conference on Mobile and Ubiquitous Systems: Networks and Services, 2006.
[32] A. Sabelfeld, A.C. Myers, Language-based information-flow security, IEEE J. Sel. Areas in Commun. 21 (2003) 5–19.
[33] F.L. Tiplea, D.C. Marinescu, Structural soundness of workflow nets is decidable, Inform. Process. Lett. 96 (2) (2005) 54–58.
[34] M. Wang, R. Kotagiri, J. Chen, Trust-based robust scheduling and runtime adaptation of scientific workflow, Concurr. Comput.: Pract. Exper., in press.
[35] P. Yang, Z. Yang, S. Lu, Formal modeling and analysis of scientific workflows using hierarchical state machines, in: Proc. of the International Workshop on Scientific Workflows and Business Workflow Standards in e-Science, 2007, pp. 619–626.
[36] U. Yildiz, C. Godart, Information flow control with decentralized service compositions, in: IEEE International Conference on Web Services, 2007, pp. 9–17.
[37] Y. Zhao, M. Hategan, B. Clifford, I.T. Foster, G.V. Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, M. Wilde, Swift: Fast, reliable, loosely coupled parallel computation, in: Proc. of the IEEE International Workshop on Scientific Workflows (SWF 2007), 2007, pp. 199–206.